university of groningen

faculty of mathematics and natural sciences

Rationally: A viewpoint-based decision documenting tool for Microsoft Visio

Bachelor thesis

Authors: Ronald Kruizinga & Ruben Scheedler
Supervisors: Paris Avgeriou & Christian Manteuffel

July 8, 2016

Abstract

Knowledge management plays a major role in global software development nowadays. However, due to a lack of tools and the limited scopes of the tools that do exist, there is no proper tool that allows for the documentation of architectural decisions in the embedded system industry. We developed an add-in for Microsoft Visio with the aim to solve this problem, using the flexibility and large supply of components offered by Visio, allowing architects to document decisions from various viewpoints.

During the creation of this add-in, we had to face several challenges: designing procedures for layout management, implementing undo and redo behaviour, and interacting with the Visio API in an object-oriented fashion. Although the add-in is fully functional in what it was required to do, further extensions are still possible.

Contents

1 Introduction

In the area of embedded systems, decisions are often shared in unstructured and informal manners, such as emails or meetings. This increases the risk of losing key information regarding a decision and makes a decision's rationale hard to uncover. If decisions are not shared at all, the reasoning behind them is only known to the decisions's makers and cannot easily be uncovered by an outsider. In addition, it can be a challenge to correctly capture a decision inside documentation due to a lack of proper tools for this application. These tools tend to be limited in scope or are not correctly embedded in the design process [1].

Since sharing decisions in an efficient way can be very useful to increase documentation and improve understanding of the project, the goal of the Rationally project is to develop a tool that can be easily used for documenting and sharing decisions.

The tool will be developed for the embedded systems industry. This industry concerns systems that are functioning independently within a larger appliance, such as a dishwasher. Those systems all require design documents explaining them and these should therefore contain system-specific components. This can complicate matters, since not all tools support all the different kinds of components that can be found in an embedded system.

The tool is developed as an add-in for Microsoft Visio[1]. To support flexibility in the actual components of the decision documentation, it was decided that the Rationally application would be developed for Microsoft Visio. Visio has many components available for designing a system and it can also easily be extended with custom component sets. In addition to flexibility, many companies already use the Microsoft Office suite or Microsoft Visio. This means it would be easy to start using Rationally, due to the applications being well known to the user. In addition, Visio is greatly extensible, in that it offers an API based on which extensions called add-ins can be integrated.

Rationally is designed to be intuitive for users in many different disciplines of software engineering. It could work together with a lightweight server storing the decisions for use in a distributed environment. The server could also allow finding and storing relations between decisions. Finally, it uses a viewpoint-based approach to documentation.

We will start off by introducing the background of decision making and documentation. Then, we will discuss the requirements for the Rationally application. Next, we will discuss design and architecture of the application. We will finish our thesis by presenting our evaluation and conclusion.

2 Background

2.1 Architectural Knowledge

Knowledge management plays a major role in global software development nowadays, as stated by Clerc et al [2]. More specifically, knowledge about a system's architecture is managed. This knowledge describes the global structure of the system. Our project deals with the documentation of architectural knowledge, therefore we will first define it as follows:

[1] `https://products.office.com/en-gb/visio/flowchart-software`

Architectural Knowledge = Design Decisions + Design

This definition, proposed by Kruchten et al., states that architectural knowledge consists of two parts: *Design Decisions* and *Design* [3]. Architectural knowledge (AK) comes in different forms. For example, it was pointed out by Farenhorst and De Boer that AK can be either explicit or implicit (tacit) and it can be application-generic or application-specific [4].

Explicit AK has been explicitly communicated or written down, like the implementation of a certain design pattern in a system. Implicit AK was not explicitly communicated but rather collected through experience, for example sticking with a certain programming style. Explicit AK can be easily documented, whereas implicit AK is rather hard to document. Yet, they are both part of the architectural knowledge of a system [4]. Application-generic AK concerns knowledge, like design patterns, that is reusable is other projects. Application-specific decisions are solely applicable to a specific application.

2.2 Views on Architectural Knowledge

Research conducted by Farenhost and De Boer suggests four views on architectural knowledge that are widely used throughout the software engineering world [4]. By a view, we mean: a work product expressing the architecture of a system from the perspective of specific system concerns, as defined in ISO 42010. These four views are the pattern-centric, dynamism-centric, requirements-centric and decision-centric. We will briefly describe each of them below.

The pattern-centric view was originally used in the object-oriented area of software development and was about capturing knowledge in patterns that could be reused. Reusability is one of the advantages that come with this view on knowledge. A second advantage would be that patterns form a frame of reference for architectures to exchange architectural knowledge in an efficient manner. This view mostly captures explicit application-generic knowledge.

The dynamism-centric view finds its origin in dynamic software systems. These systems must be able to adapt their architecture and in order to do so, they rely on architectural knowledge. This means that the architetural knowledge is stored in a format that is readable for systems. The advantage of this approach is that the system can improve itself, but it also requires the system to be able to interpret architectural knowledge [4]. This approach results in the capturing of explicit application-specific architectural knowledge.

The requirements-centric view focuses on closing the gap between requirements and architecture. It attempts to let requirements and architecture be co-developed, instead of making one after the other. The idea behind this is that for some requirements, possible solutions need to be known [4]. This view results in the documentation of explicit application-specific architectural knowledge.

The final view that we will discuss in this section is the decision-centric view. This view attempts to not only capture the end result of a system's architecture, but also the design rationale behind it. The concerns for making certain decisions are not only considered, but also documented for the future, to make sure they are not forgotten. The decision-centric view captures implicit application-specific knowledge in the form of concerns, but also explicit application-specific knowl-

edge: the resulting decisions themselves. The rationale behind a decision consists of various parts. For example, it consists of picking between multiple alternatives, and considering certain forces (concerns) that affect the feasibilty of the alternatives. An approach for looking at a certain aspect of the rationale behind a design decision, we will call a viewpoint. In the following section, we will take a closer look at the decision-centric view by explaining different viewpoints on a decision.

2.3 Architecture Viewpoints

According to Van Heesch, documenting decisions in a viewpoint-based manner is a promising approach [5]. ISO 42010 defines a viewpoint as a work product that establishing the conventions for the construction, interpretation and use of architecture views to frame specific system concerns. In this article, a new framework for documenting decisions is proposed, based on four different viewpoints: a Decision Detail viewpoint, a Decision Relationship viewpoint, a Decision Chronology viewpoint and a Decision Stakeholder Involvement viewpoint. In another article, Van Heesch introduces a fifth viewpoint, the Forces viewpoint [6]. The goal of this framework is to give concrete advice on how to construct a set of consistent architecture decision views by satisfying several concerns related to decision documentation. It does so by looking at a problem that needs to be solved from the perspectives of several different stakeholders, basing the viewpoints on groups of stakeholders.

The Decision Detail viewpoint gives detailed information about single decisions. This viewpoint is the closest to traditional methods of documenting decisions. It documents descriptions of the problem, alternative solutions and arguments for the decision. This viewpoint is relevant to most stakeholders, but mainly to reviewers and architects.

The Decision Relationship viewpoint makes relationships between architecture decisions explicit. Using this viewpoint, relations such as causation and depency can be documented. This viewpoint is relevant to architects, domain experts and reviewers.

The Decision Chronological viewpoint shows the evolution of architecture decisions over time. This viewpoint shows iterations of decisions and it's endstate. Iterations are specified by version numbers and/or dates. This viewpoint is mostly relevant to architects, reviewers, and new project members.

The Decision Stakeholder Involvement viewpoint shows the responsibilities of relevant stakeholders in the decision-making process. This viewpoint shows the decisions, the actions taken and involved stakeholder(s) within a single iteration. The point of this viewpoint is to personalise knowledge; to document who knows what. Typical stakeholders for this viewpoint are reviewers, architects and managers.

Views using the Decision Forces viewpoint make the relationships between architectural decisions explicit, in addition to the forces that influenced the architect when making the decisions out of multiple alternatives. Van Heersch defines a force as "any aspect of an architectural problem arising in the system or its environment, to be considered when choosing among the available decision alternatives [6]." The forces can easily be presented using a table, providing information about how a certain force is satisfied by a decision.

2.4 Alternative Products

There have only been a few tools that fulfill a similar role to the Rationally add-in. We will discuss them here and also the drawbacks and benefits provided by each tool. The tools we will discuss are Microsoft Visio and Decision Architect.

Microsoft Visio allows its user to easily create flowcharts and spreadsheets using provided shapes and functionality (Figure 1). The benefits of using Visio include the fact that the tool is open to any approach the user wishes to take. However, this also means that native support for documentation is not as prevalent. The extensibility of the application also allows for a lot of options. The user can easily download or create shapes or templates, which can really improve the workflow. The user can also create their own add-ins, as we have done with Rationally. This makes Visio really powerful for a user that knows how to use it.

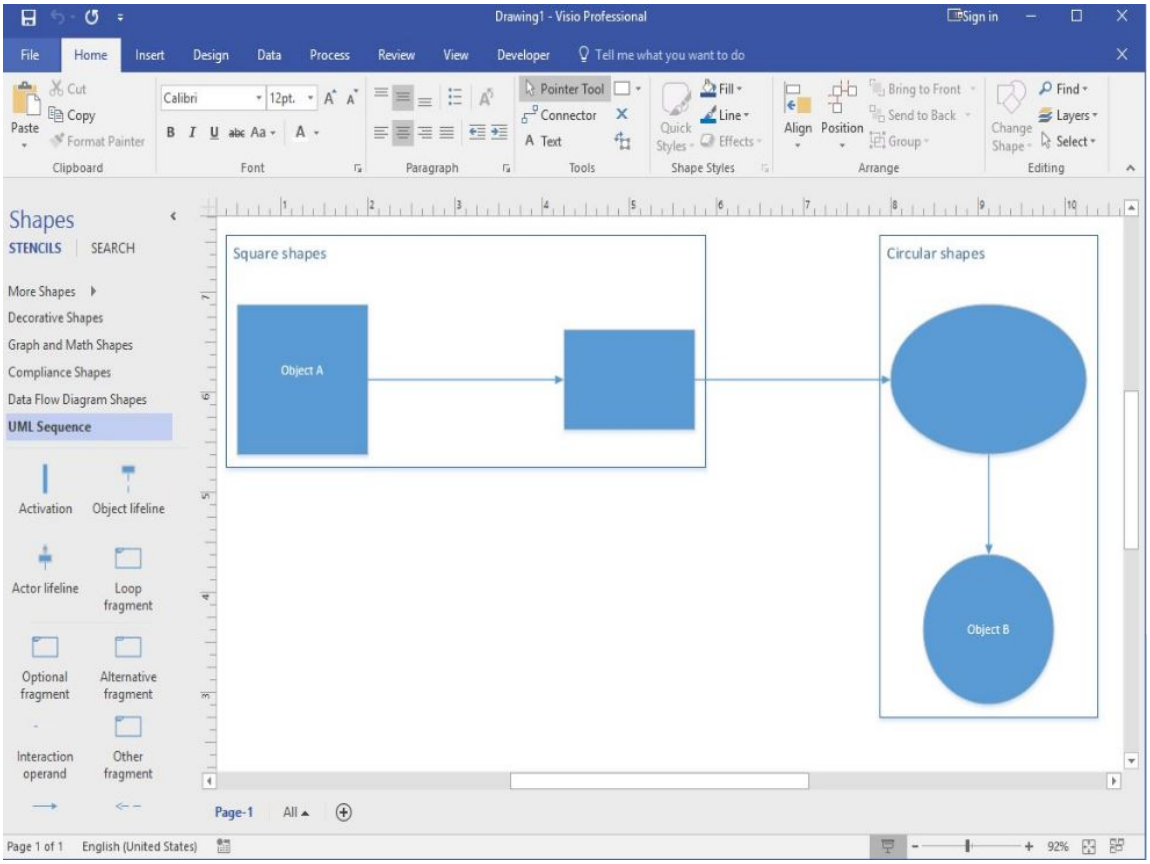

Figure 1: Example usage of Visio.

Like Rationally, Decision Architect is a viewpoint based documentation tool [7], using the viewpoints described by Van Heersch et al [5]. Studies have shown that it is able to capture all aspects

of a decision and is able to improve the quality of the documentation. However, due to being an add-in for Enterprise Architect, usage of the tool is limited to companies using Enterprise Architect.

Glossary

In this section, we will introduce a few terms that are frequently used throughout this document. Terms are provided together with their definition.

Term	**Definition**
Alternative	Candidate solution, represented by a shape or container.
Container	Shape that can contain other shapes.
Field	Text area found on the sheet.
Force	Concern or force influencing a decision, represented in a table.
Shape	Visible object in Visio that is completely customisable.
Shapesheet	Table-like object that stores a shape's properties.
Sheet	Area in the Visio application where all actions take place.
Template	Reusable Visio sheet with provided shapes.
View	*See Sheet*
Viewpoint	Different approach to look at a decision.

3 Requirements

The add-in will provide Visio users with a way to accurately document and share architectural decisions. The format allows the user to add or remove different components to the sheet, such as displaying alternatives, concerns or related documents. The goal of the add-in is to capture all five viewpoints in a single application. It should allow architects to use it in an easy and convenient way by adding context to many of the actions provided by Visio.

In this section, we will enumerate and describe the requirements that were given for the Rationally add-in. The requirements were received through an example image (Figure 2 & 3) that showed how the application was approximately supposed to look, complemented by the requirements that we discovered during meetings. The requirements are defined using the MoSCoW-method, which means they are prioritised according to the following categories[2]: *Must have, Should have, Could have* and *Won't have*. All requirements are preceded by a description.

[2] `https://en.wikipedia.org/wiki/MoSCoW_method`

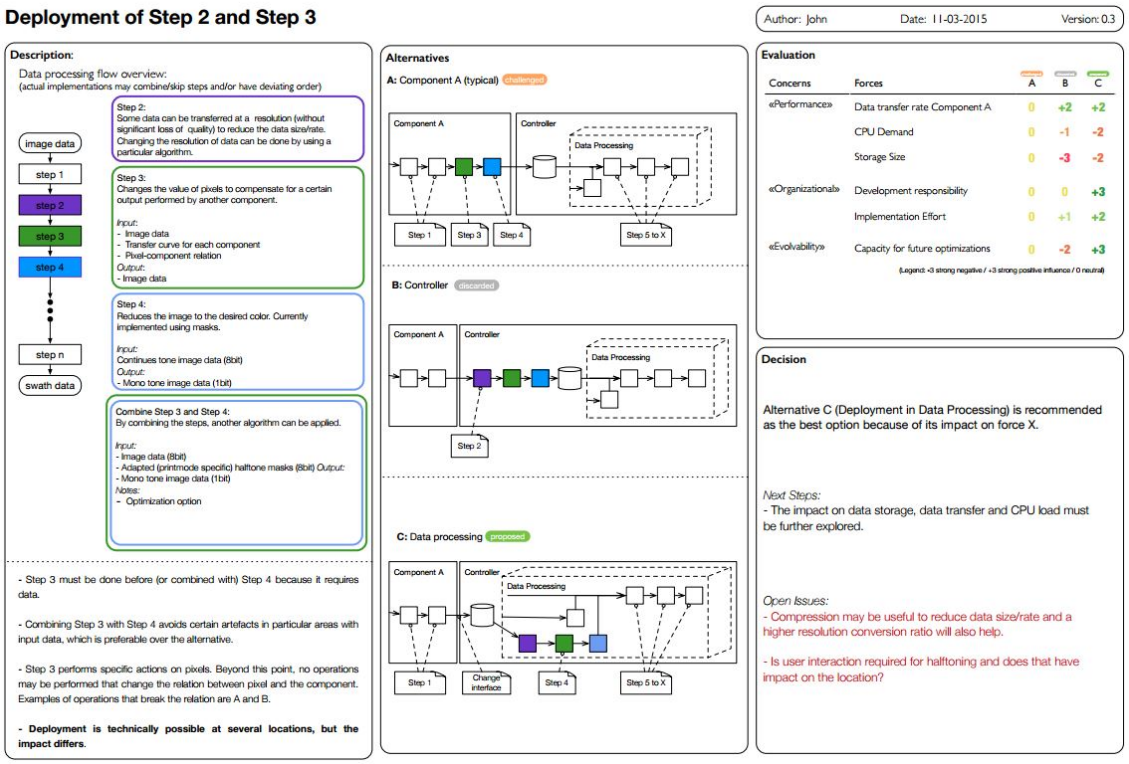

Figure 2: Provided requirements document (top half)

R1: Decision Information

Decisions can still change often, which means it is necessary to have some information provided to the user regarding the decisions shown here.
The information that is necessary to reach this goal is: the name of the decision, the author of the sheet, the version of the decision and the date on which the decision was taken. Using this information, the user will be able to easily track the decision.

1. The name of the decision must be shown at the top of the view.
2. It must be possible to enter the name of the decision at document creation.
3. The name field should be optionally deletable.
4. The author of the sheet must be shown at the top of the view.
5. It must be possible to enter the author of the decision at document creation.
6. The author field should be optionally deletable.
7. The date of creation of the decision must be shown at the top of the view.
8. It must be possible to enter the date of creation at document creation.
9. The date of creation field should be optionally deletable.
10. The version number of the decision must be shown at the top of the view.

11. It must be possible to enter the version of the decision at document creation.
12. The version number field should be optionally deletable.

R2: Description

One of the important parts of a decision is the description, which is a brief problem statement that conveys the context of a decision. This can possibly be done with flowcharts of the implementation of this decision. Therefore, there must be a component in which the user can provide this information. No explicit functionality is added to the component, since the user must be completely free to decide how to provide the description.

1. The description section must be deletable at all times.
2. It must be possible to add a description section at all times.
3. The description section should be available by default on the template.

R3: Alternatives

In the decision-making process, architects usually consider multiple alternative solutions that could solve the problem. Eventually, one of these alternative solutions is chosen.
Documenting not only the chosen solution but also the considered solutions is important, because it provides important rationale that is particularly helpful during maintenance and evolution, e.g., whether a certain solution was considered before or whether we need to investigate it. It can also be useful to know why a solution was not selected and whether the same reasons still apply.
This way the user can always see which alternatives have already been discussed and which might still provide a new approach.

1. The alternatives section must be deletable at all times.
2. It must be possible to add the alternatives section at all times.
3. The alternatives section should be available by default on the template.
4. The must only be one instance of the alternatives list at a time.
5. All updates to the list of alternatives must update the forces table (R3).
6. Alternatives must be addable using a context menu.
7. Alternatives must be deletable using a context menu.
8. Alternatives must be editable using a context menu.
9. Alternatives must be reorderable using a context menu.
10. The state of an alternative must changable using a context menu.

R4: Evaluation

When making a decision, there are multiple forces (as described by Van Heersch as the forces viewpoint [6]) and concerns, such as performance or ease-of-use, influencing the decision. Therefore, a forces overview is provided on the sheet, allowing the user to quickly enter or see the benefits of certain decisions in relation to the alternatives.

1. The forces section must be deletable at all times.

2. It must be possible to add the forces section at all times.
3. The forces section should be available by default on the template.
4. The must only be one instance of the forces table at a time.
5. Forces must be addable using a context menu.
6. Forces must be deletable using a context menu.
7. Forces must be editable using a context menu.
8. Forces must be reorderable using a context menu.
9. Every row in the forces table must contain a name, a description and a value for each alternative.
10. The user must be able to change a value at any time.
11. Changing a value must automatically update the total.
12. Colums must be ordered according to the order of the alternatives.
13. Value cells should change colour depending of the value.

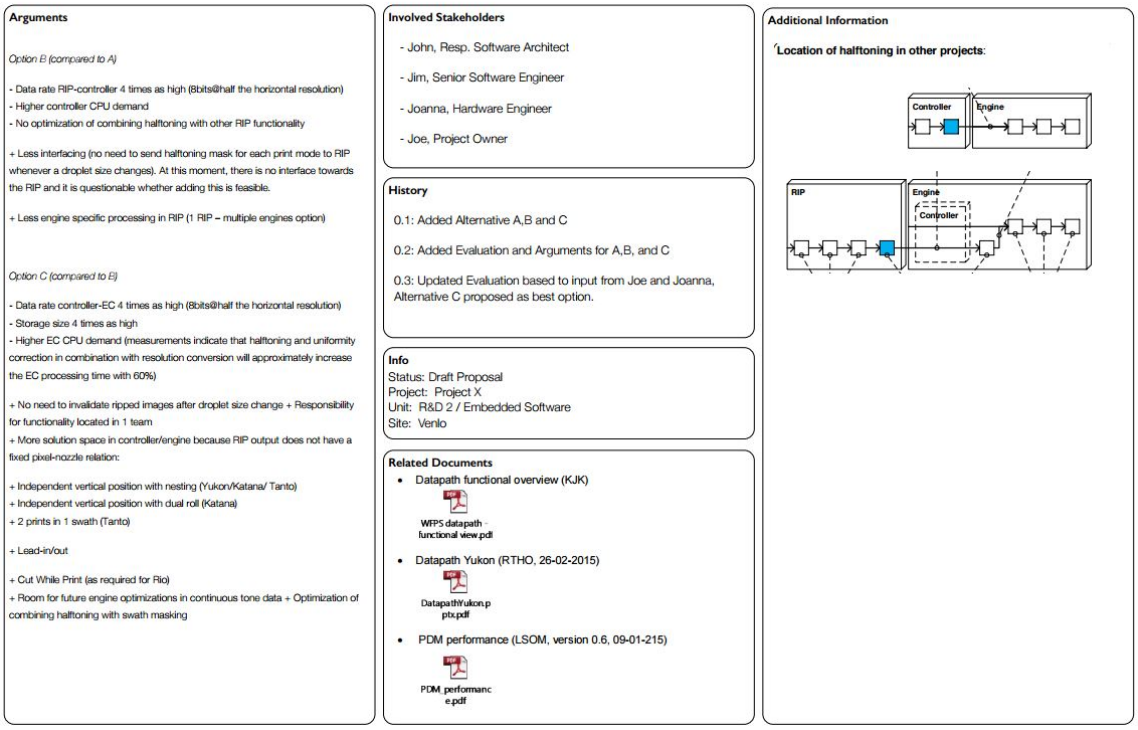

Figure 3: Provided requirements document (bottom half)

R5: Related Documents

No decision is taken in a vacuum, which means that there are often documents or webpages related to the decision, containing information or argumentation. These files and links are therefore also provided on the page in order to easily view and access this information.

1. The documents section must be deletable at all times.
2. It must be possible to add the documents section at all times.
3. The documents section should be available by default on the template.
4. The documents section must support both links and filepaths.
5. Documents must be addable using a context menu.
6. Documents must be deletable using a context menu.
7. Documents must be editable using a context menu.
8. Documents must be reorderable using a context menu.
9. There must only be one instance of the documents list at a time.
10. Documents must show the full path or url to the document.
11. Documents must be openable by doubleclicking.
12. Documents must be openable by ctrl+clicking.

R6: Arguments

Since the alternatives have already been provided, it is useful to also see a descriptive reason regarding why they were or were not chosen as the correct approach. Again, no explicit functionality is added to this component, so that the user is free to decide in what way he wants to provide this information.

1. The arguments section must be deletable at all times.
2. It must be possible to add the arguments section at all times.
3. The arguments section should be available by default on the template.

R7: History

Since documents often get updated after creation, a history component, similar to a changelog, is provided. In the component, the user can describe what changes were done when.

1. The history section must be deletable at all times.
2. It must be possible to add the history section at all times.
3. The history section should be available by default on the template.

R8: Involved Stakeholders

In the end, knowledge remains in the heads of the people involved with a decision. It is generally not possible or viable to document all knowledge, which means that it is very useful to know who was involved with the decision. This way, knowledge and rationale can easily be traced back to a person.

1. The stakeholders section must be deletable at all times.
2. It must be possible to add the stakeholders section at all times.
3. The stakeholders section should be available by default on the template.

R9: General

Besides the context specific requirements, there are also a few requirements for the entire application and its codebase.

1. All components of the template must fit on an A3 when printed.
2. All operations in the application must support undo and redo.
3. The code for the application must be written in C#.
4. The application must support saving of the sheet.
5. The application must support loading of a sheet.

R10: Non-functional Requirements

In addition to the functional requirements, there are also non-functional requirements that had to be taken into account. The userfriendliness requirements were amongst the most important requirements of the application, since they concern the visual components. The other non-functional requirements had a lower priority.

1. Extensibility: The codebase must be written in a way that allows for extensions.
2. Maintainability: The codebase must be well documented using comments and self-documenting code.
3. Maintainability: The codebase should be designed in a modular way, to allow for changes that do not propegate endlessly.
4. Maintainability: The codebase must adhere to Resharper[3] standards.
5. Serviceability: The add-in should be easy to install.
6. Usability: Operations should not take more than 1 second.
7. Usability: When more than three alternatives are added, a warning regarding the layout should be shown to the user.
8. Userfriendliness: The application must be intuitive and easy to use.
9. Userfriendliness: The application should use the least amount of dialog boxes as possible.
10. Userfriendliness: The state of an alternative should have a recognisable colour.

[3] `https://www.jetbrains.com/resharper/`

4 Product Functionality

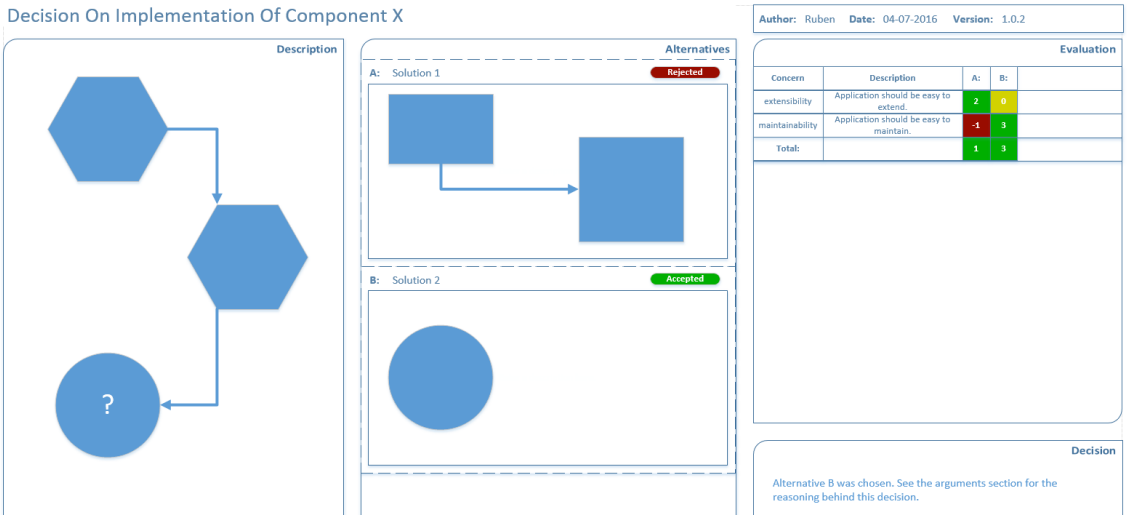

Figure 4: Screenshot of the top half of the Rationally sheet in Visio.

In Figure 4, the top half of the Rationally sheet is displayed, containing example data for a decision on the implementation of a component X. At the top of the sheet the title of the decision is displayed and next to it the author, date and version of the decision.

Below that, in the top left area, we find the *Description* area, which is part of the view of the decision detail viewpoint. This area will contain a description of the context of the decision that is being documented.

In the top middle of the sheet, the area for *Alternatives* can be found. This area is also part of the decision detail view and its purpose is to document the candidate solutions or *Alternatives* of a decision. The alternatives can easily be added, removed and reordered as a whole using context menus, which holds true for all containers with a list structure (i.e. *Alternatives, Evaluation* and *Related Documents*). The state of each alternative can also be changed using its context menu.

On the right side, we find two areas in the top half of the sheet. The second one is the *Decision* area, which is intended to document the details of the chosen alternative and its consequences. Above that, we find an area with a lot of automated functionality. Firstly, the list functions described for alternatives are available here, but there is also automated behaviour that has to do with the content of this area.

The *Evaluation* area is structured as a table where architects can add concerns for the decisions (as rows). Every row automatically receives a column cell for every alternative present in the *Alternatives* area. The architect can enter values here, indicating how well an alternative takes a concern into account. The final row of the table contains the totals for each alternative, which are

calculated automatically. The background colour of the cells is automatically determined by their value (positive becomes green, zero becomes yellow, and negative becomes red). This concludes the main functionality of the top half of the sheet.

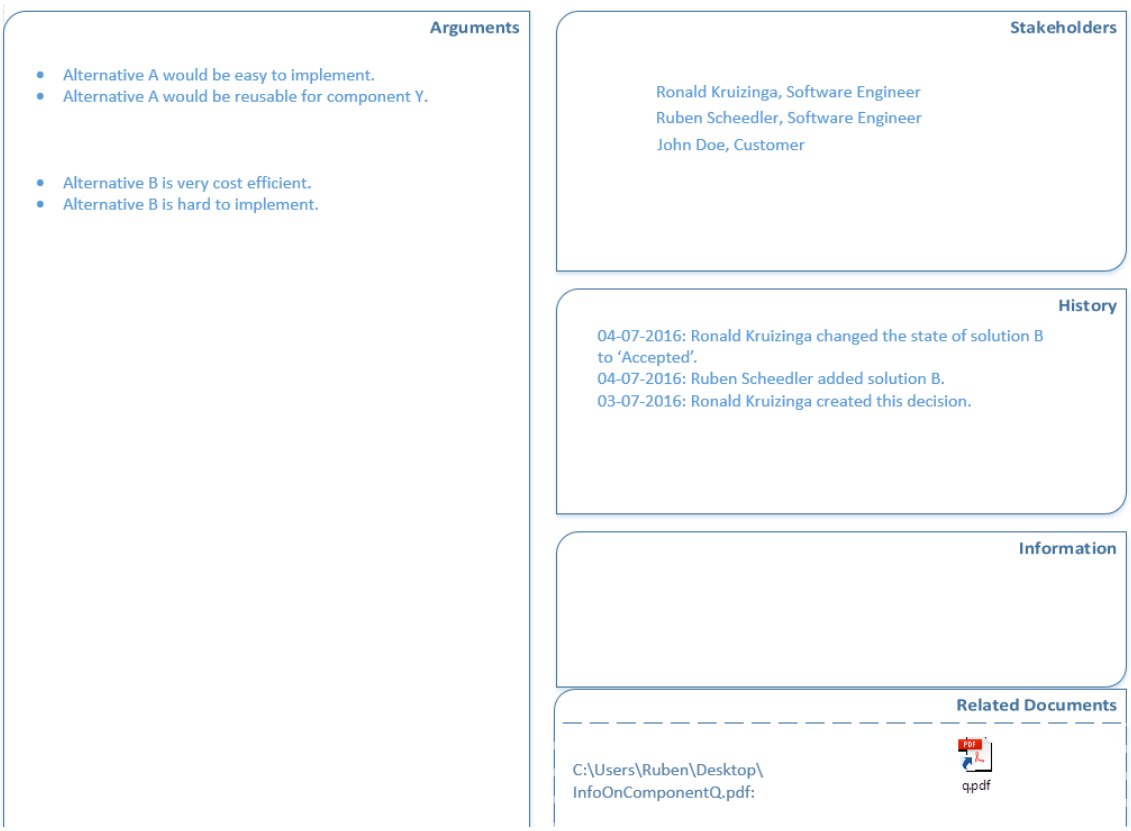

Figure 5: Screenshot of the bottom half of the Rationally sheet in Visio.

The bottom half of the sheet is more free, in a way that its containers have less constraints regarding content (Figure 5). Only the *Related Documents* area is being programatically managed (that is, its content is stored in the model of our application). This area is dedicated to containing documents that are relevant for the making of the decision. A related document can either be a file, that will be included in the Visio file, or a hyperlink to a source on the internet. Both are accompanied by a hyperlink element, a text element containing the path/url and a title. Related documents might be used for documenting the relation viewpoint, by referencing other decisions for example.

Furthermore, there is an area for the stakeholders of a decision. This area's function is simply to list the stakeholders of a decision and it implements the stakeholder view.

Underneath the stakeholders area we find the history area, that captures the history of the decision sheet (e.g. the changes made to it). This helps with documenting the chronological viewpoint.

The other areas that are available are the *Arguments* area, which captures the arguments pro and contra the alternatives present in the sheet (which is part of the decision detail viewpoint), and the *information* and *additional information* areas, which are respectively intended to contain information about the decision and additional (technical) context for the decision.

5 Architecture

In this section, we will provide information on several topics related to the architecture of Rationally. First, we will highlight and explain some components of the visio API that the Rationally plug-in will interact with. Secondly, we will explain some important decisions we had to make during the development of the add-in. We will do so by explaining a problem, followed by candidate solutions for the problem. After that, we will describe concerns that influence our decision and how well each solution handles the concern. Finally, we will describe arguments for the different alternatives.

5.1 Visio API components

Stencils, Masters, Shapes

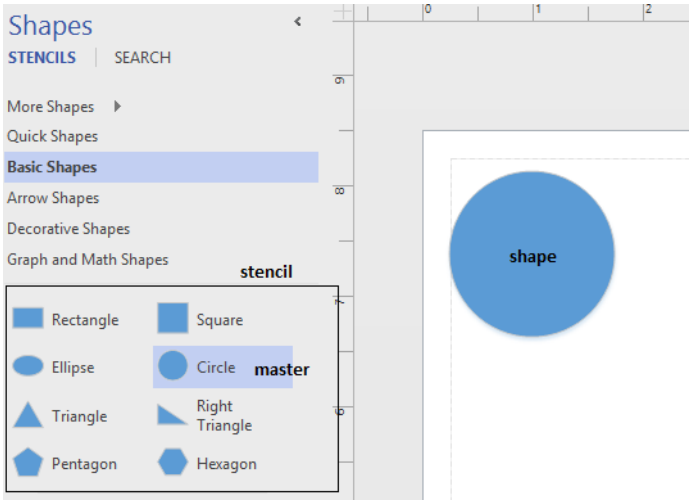

Figure 6: Visual explanation of a stencil, a master and a shape.

Visio offers three major components to the user to fill the page of a Visio document. The first one is a stencil. A stencil can be opened by the user and offers him a collection of shape types. These shape types are called masters. They are responsible for instantiating shapes on the page. The shapes are instances of a master, dropped on the sheet.

ShapeSheet

The ShapeSheet object is an important component in our project, because it defines most of a shape's properties. A ShapeSheet is a table-like object and each shape in Visio has its own. It consists of a fixed set of defined sections (Figure 7). Each section has the form of a table, with rows and columns. Every section is responsible for a certain aspect of a shape. The *Actions* section, for example, defines context menu options and the *Protection* section contains rows that determine whether certain actions are allowed on a shape, like moving and resizing. Each cell contains either a value (and an internal unit of that value) or a formula, defining a procedure. The ShapeSheet as a whole resembles an Excel sheet, without the possiblity to style the cells.

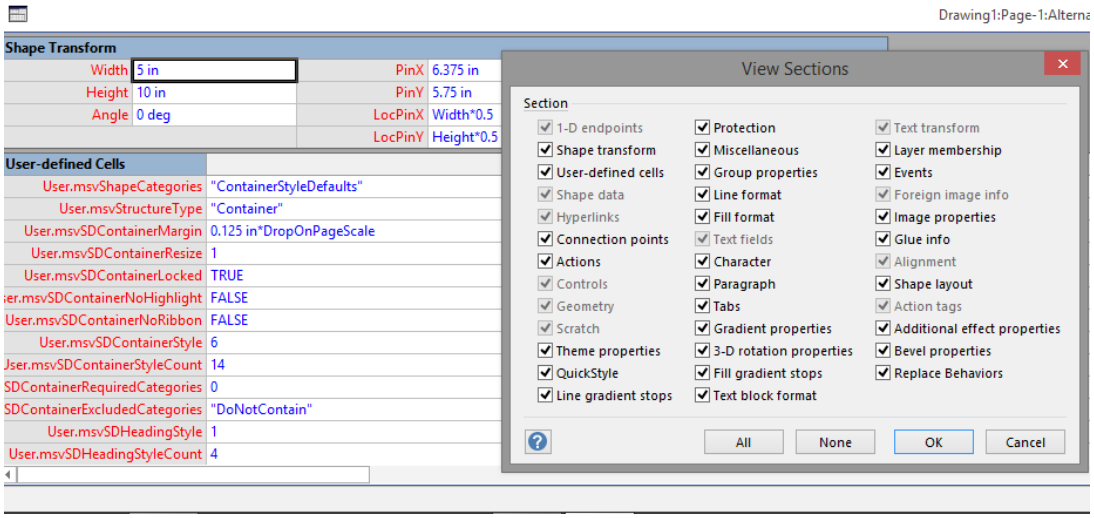

Figure 7: Example of a ShapeSheet, accompanied by an overview of sections.

When Rationally needs to modify a certain property of a shape, it will do so by modifying its ShapeSheet. The Visio API offers several methods to change a shape's ShapeSheet. Below, some example statements are given that interact with a ShapeSheet.

```
1 //Store the width of RShape, given in its internal units.
2 double width = RShape.CellsU["Width"].ResultIU;
3
4 //Set the center of RShape in x to 5 inch.
5 RShape.CellsU["pinX"].Result[VisUnitCodes.visInches] = 5;
6
7 //Define a context menu option for RShape, that fires a marker event
      when clicked.
8 RShape.CellsU["Actions.CustomAction.Action"].Formula = "
      QUEUEMARKEREVENT(\"customActionFired\")";
9 RShape.CellsU["Actions.CustomAction.Menu"].Formula = "click me";
```

Visio Events

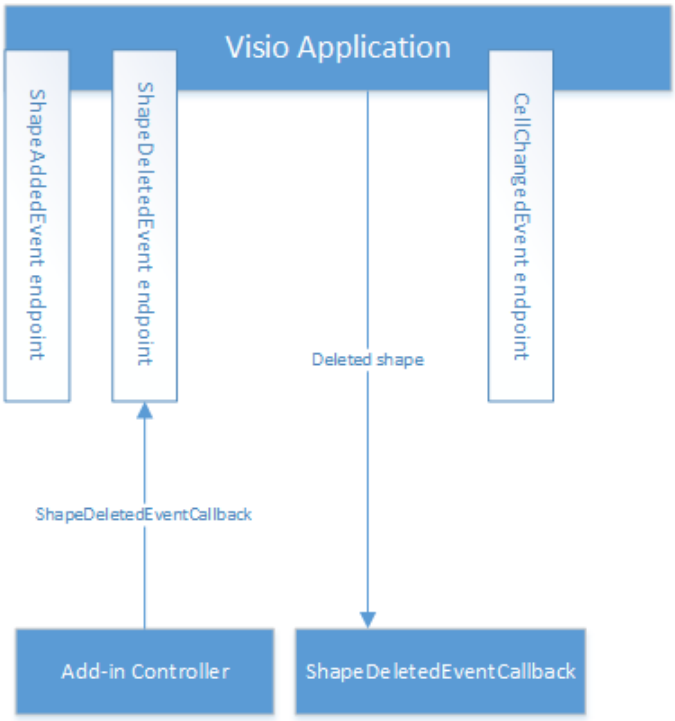

Figure 8: Overview of event handling in a Visio add-in.

The Visio API makes it possible to not only create add-ins, but also let them tune in to events generated by the application. Figure 8 shows an overview of event handling in a visio add-in. The visio Application[4] object is the object our plug-in interacts with. We will describe some parts of its interface before explaining its role in event handling.

1. *ActivePage*: A reference to the page that the user is working on; is also the container for all the shapes of the page.
2. *ActiveWindow*: A reference to the Visio window that is active (used to find the shapes that are selected by a user).
3. *CustomMenus*: Gets an object containing the customly added menus.
4. *Documents*: Contains a list of documents (e.g. stencils that are opened by the user) in use by the application.
5. *MarkerEvent*: Endpoint to register event listeners for a marker event. The Application object has such an endpoint for all events.

[4] `https://msdn.microsoft.com/en-us/library/office/ff766485.aspx`

As seen above, the Visio Application object has a seperate endpoint for each type of event, implemented as a list property. Add-ins can register event listeners at these endpoints. The listeners are procedures that conform to a specific signature, depending on the event type. When an event is fired in the application, all registered event listeners are called and passed the appropriate parameters. To register an event handler for a marker event, one would need to define the following.

```
1 //register the handler with the Application object
2 Application.MarkerEvent += Application_MarkerEvent;
3
4 //defintion of the event listener method
5 private void Application_MarkerEvent(Application application, int
      sequence, string context)
6 {
7     //handle marker event
8 }
```

It is important to note that the parameters are not event objects, but rather view components or primitives. The application object thus hides the implementation of the event firing and the event classes. This results in the constraint for add-ins of not being able to fire these events, only listen to them. The only exception to this rule is the *marker event*, that can be fired using the VBA function *QUEUEMARKEREVENT(context:string)*.

5.2 Decisions

In this section some major decisions that were made during the process are described and argumented. They are accompanied by an identifier (D[number]) to allow referencing to them in this document. For each decision, we describe the context of the problem/requirement, the possible ways to solve/implement it and the forces influencing it.

D0 - Wrapper Layer Around the Visio API

Context
The context of this problem is the interaction with a shape's ShapeSheet, as described in section 5.1.
Problem
The Visio ShapeSheet treats every shape in the same way. Shapes with no actions offer the same interface as Shapes with many actions. The same is true for containers, that offer the same interface as non-container shapes. Not only do all shapes offer the same interface, all rows in their ShapeSheet hide their internal unit. This means that the API offers no type safety. The problem with this lack of type safety and uncertainty of the actual available interface, is that the developer needs a lot of additional logic (e.g. existance checks, type checks) to interact with the ShapeSheet.

Candidate Solutions

1. **(rejected)** *S0: Direct interaction with the Visio API* - We do not build any code around the visio API and we always interact with it directly.
2. **(rejected)** *S1: Maintain a parallel model and update when needed* - We design a model structure similar to the ShapeSheet and interact with it, instead of the actual ShapeSheet. We update the actual ShapeSheet in an observer-like fashion.
3. **(accepted)** *S2: Create a wrapper layer around the visio API* - We create an API around the ShapeSheet API that visio offers that offers more type safety and similar constraints than the actual API by validating values and fields before updating the ShapeSheet.

Forces

Concern	**Description**	**S0**	**S1**	**S2**
extendability	New shapes must be easy to integrate.	3	2	2
usability	The solution should be easy to use.	0	3	3
complexity	The solution should not be complex when implemented.	-1	0	3
flexibility	The solution must be able to handle changes in the model.	-3	2	2

Table 1: Forces that play a role when deciding upon a solution for D0.

Arguments
S0 requires very little short-term work because no extra code need to be written. The API is also very flexible, in a way that ShapeSheet cells accept various units, so the "type" of cells remains flexible even on runtime. However, this also results in the aforementioned lack of type safety. The uncertain interface of shapes can be solved by checking for the existance of a field at every read and write attempt. This would, however, make the code more complex.

S1 solves both the type safety and the uncertain interface problems. However, it requires a lot of short-term work, because the complete ShapeSheet needs be mimicked. On top of that, some way of synchronizing the ShapeSheet with the new model would be required, which also comes with a lot a work and complexity.

S2 also solves both described problems and does this in simpler way. The multi-layered inheritance tree allows for varying in interface of different components, and the type-safety can be guaranteed by defining our own read/write interface for the ShapeSheet data. This interface can also abstract safety-checks related to internal units and existance of cells.

D1 - Designing Layout Managers

Context
Visio shows a page,on which shapes can be positioned and customized, to the user. The shapes are also ordered in a composite-fashioned way. Visio offers various sets of simple shapes, that is, they

are single-layer singular objects. Creating and maintaining a hierarchical structure is left to the user, and so is the positioning of the shapes. Only aligning tools are offered.

Problem
Our project, on the other hand, offers a set of relatively complex multi-component multi-layered shapes. The problem that occurs is that we have to programmatically structure and position shapes. Because this is generally the responsibility of the user, the visio API does not offer any layout managing.

Candidate Solutions

1. **(rejected)** *S0: Make our components responsible for layout mananing their content.* - We implement the layout managing in each component that requires it, and tailor that code to the specific component.

2. **(accepted)** *S1: Create layout managers.* - We create generic layout managers that can manage the layout of any component. Each manager implements its own type of layout managing, for example by stacking all elements vertically, or placing them in a horizontal line. By using a composition of these generic managers, more complex layouts can be defined.

Forces

Concern	**Description**	**S0**	**S1**
extendability	New shapes must be easy to support.	0	3
usability	The solution should be easy to use.	-1	2
complexity	The solution should not be complex when implemented.	-1	3
flexibility	The solution must be able to handle changes in the model.	-2	3

Table 2: Forces that play a role when deciding upon a solution for D1.

Arguments
S0 is a quick solution. Because every component is responsible for its own layout management, it allows for very tailored code. Using component-specific constants, a layout is easily created. However, the constants are magic numbers which are not flexible at all. The code tends to become complex with a lot of long mathmatical additions and subtractions of constants and factors.
This solution neither offers any reusability, and is not easy to extend.

S1 on the other hand, requires quite some work to create the layout managers. They should be incredibly flexible, making their creation relatively hard. Once they are created, however, they bring extensibility and usability, in the sense that you only need to define a certain manager, instead of writing the layout code for every component.

D2 - Handling Custom Styling of Rationally Components

Context
The Rationally add-in generates (groups of) shapes for the user to make the process of decision documentation faster and more structured. Which shapes are displayed on the view to the user depends on the model that is maintained in the add-in. A model change leads to a repaint of the view, to make the view up to date.

Problem
A repaint of the model generally only considers the state of the model and nothing else. In Visio, it is possible for the user to style components, even the ones generated by the add-in. The problem that would occur is that the style added by the user to a component generated by Rationally is discarded when a repaint takes place, because the representation of the model data (the styled shape) is not part of the model.

Candidate Solutions

1. **(rejected)** *S0: Disable styling on rationally components.* - Components that are generated by the add-in can not be styled in any way. No styling can be lost this way.
2. **(rejected)** *S1: Include the style of components in the model.* - Besides the data of model components, we store the styling of the component that represents that data. We would fire events at every styling action and update the model accordingly.
3. **(accepted)** *S2: Maintain a tree of components generated by our add-in.* - Parallel to the model, we maintain a state of the view (i.e. the page the user is working on) in which we store the components generated by Rationally. This state maps model components to view components (shapes), so that model updates can modify a view component instead of overwrite it.

Forces

Concern	Description	S0	S1	S2
extendability	New add-in parts must be easy to integrate.	3	-1	1
functionality	The solution should not affect the available functionality.	-3	3	3
complexity	The solution should not be complex when implemented.	2	-2	1
flexibility	The solution must be able to handle changes in the form of the model.	3	-1	0
ease of implementation	The solution must take as little time as possible to implement.	2	-2	0

Table 3: Forces that play a role when deciding upon a solution for D2.

Arguments
S0 has many advantages, as can been seen in the table above. It is flexible, easy to extend, and easy to implement. However, it completely disables all styling options on rationally components, that will make up a large part of the view. Styling is too important to disable on these components, so this solution is rejected.

The second alternative, S1, does not affect any functionality but requires a lot of work. If every styling option is to be supported on components, all of them have to be included in the model and should come with event firing and handling.

The third and final available alternative, S2, also leaves all functionality intact. However, this solution is easier to implement, because only a structure containing the shapes is required instead of seperate items for each styling property (as in S1). The styling is encapsulated in the shape and all that is required is to reuse the shape on repainting. Because it offers the best functionality to the user and does not require excessive amounts of work, we decided to implement this solution.

6 Software Design and Implementation

In this section, we will describe the software design that we based our Rationally application on. It shows how the add-in is structured and what components we used. To do so, we use (UML) diagrams and explain the details of the most important components of the design.

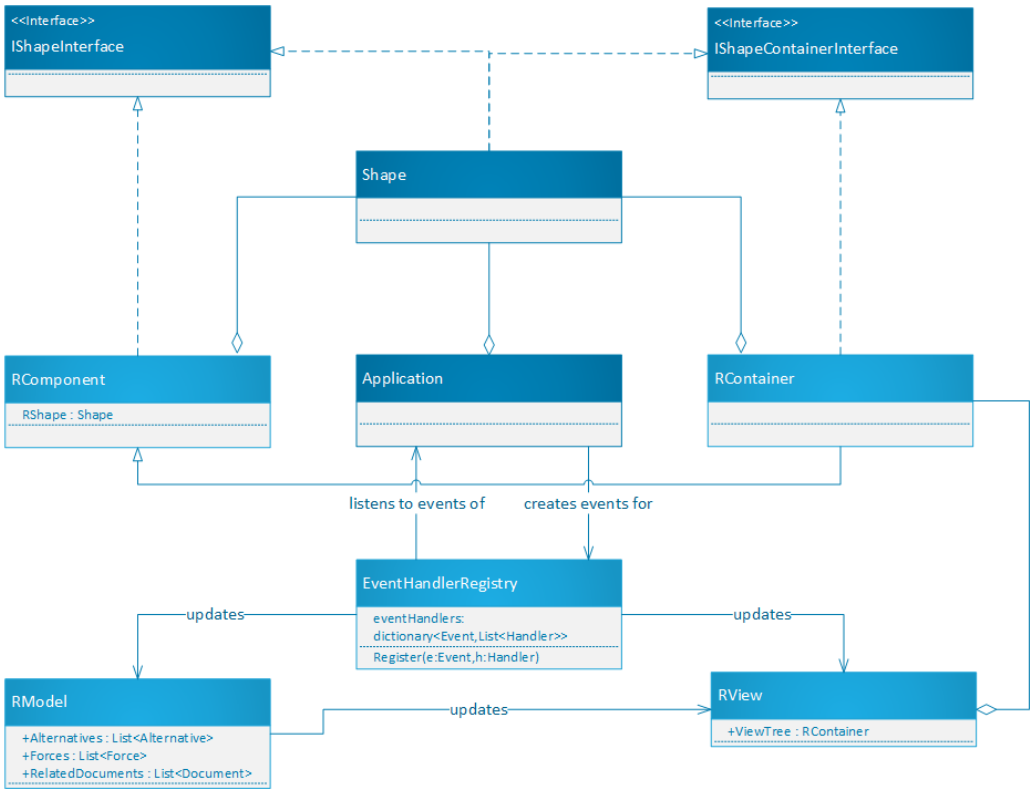

Figure 9: High level overview of the interactions of the Rationally add-in (light blue) with Visio (dark blue).

Rationally is implemented using an MVC (Model View Controller) pattern. The decision sheet represents architectural knowledge and relations between different pieces of architectural knowledge. These will be stored in a model, represented by the *RModel* class in Figure 9. It stores the alternatives, forces and related documents that the user has added in the sheet. The *RView* component will represent the visual state of the decision sheet, as it is visible to the user. This will be explained in detail in the section *View Tree*. Finally, a controller will be responsible for creating events after actions in the view and for updating the model correctly. The controller in Figure 9 is the Application object of Visio. The *EventHandlerRegistry* was built against the interface of the *Application* object and maintains a registry of procedures to execute at certain events, which are responsible for updating the model.

6.1 Composite Structure

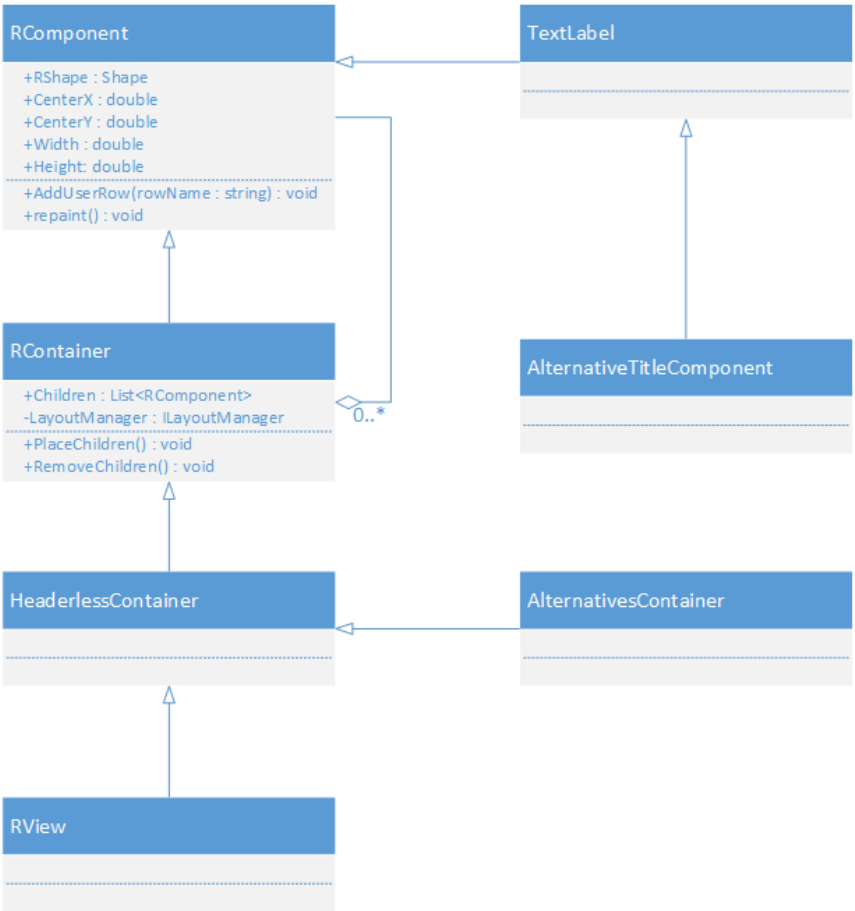

Figure 10: Use of the composite pattern in the Rationally system.

The Rationally system consists mostly of view-related components. In the *Decisions* section, it is already explained that we need to maintain a view state (see D2) and thus need a way of representing view components in our application. The Visio API already offers classes and interfaces for this (*Shape, IVShape*), but these come with some limitations. First Visio's structure does not suit an object-oriented application with a large inheritance scheme (see decision D0). Second, the *Shape* class nor the shape interface (*IVShape*) can be inherited from. The composite pattern that we introduced as a solution follows the conventionial composite pattern [8]. The pattern is implemented in the classes *RComponent*, representing the component class, and *RContainer*, representing the composite class, extending the component class (Figure 10). On top of this pattern, we have introduced various classes (e.g. HeaderlessContainer, AlternativesContainer, TextLabel) that represent different parts of the Rationally view (Figure 10). Each class comes with its own interface depending on its ShapeSheet and also has its own implementation of methods like *Repaint()* and constructors.

6.2 Layout Managers

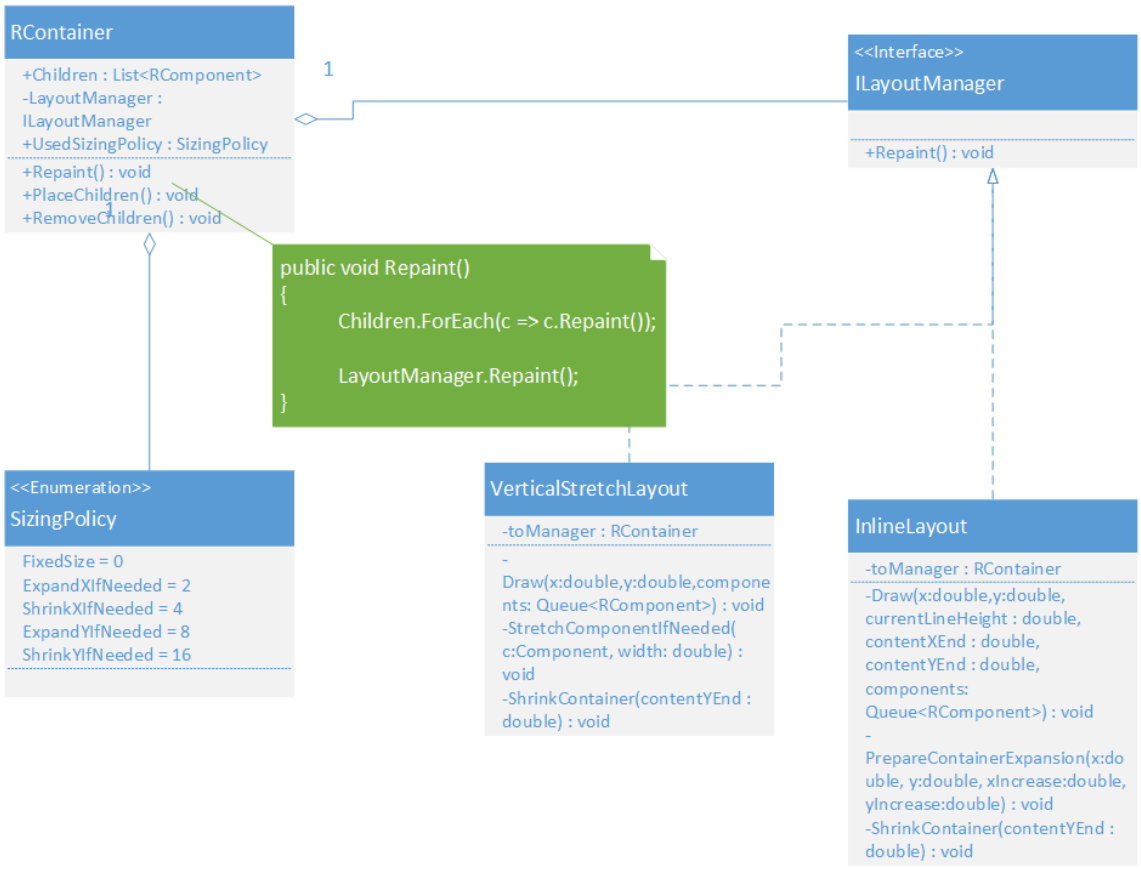

Figure 11: The implementation of layout managers.

Because our application in mostly concerned with the interaction with and the modification the view in Visio, layout managing is important in our product. The Visio API, however, does not offer any layout managers, forcing us the design our own. Layout managers are components that are responsible for determining the layout (i.e. position and size) of a group of elements, within a certain area. In our case, we designed the managers to lay out the children of a certain container.

We designed a total of two layout managers. The first, *InlineLayout* tries to place all elements horizontally next to each other. The second, *VerticalStretchLayout*, places all elements below each other in the container and stretches them up to the width of the container. As can be seen in Figure 11, every RContainer must have defined exactly one layout manager. The combination of the composite pattern described above and the two layout managers allow the creation of very complex layouts with rather simple layout managers.

Indispensable in the process of layout managing are the sizing policies that each container defines for itself. They define constraints for the layout manager of the container. Figure 11 shows the offered constraints. For both dimensions, the container can allow expansion and compression of its size. The constraints are stored in an enumeration, in which each element is implemented as a flag. This allows for fast bitwise operations on the policy field of a container. Below, two code fragments are shows. One from the vertical stretch layout performing an operation on the sizing policy of a container and one from the constructor of a container.

```
1 //stores whether the managed container is allowed to expand in the x
      direction.
2 bool expandXIfNeeded = ((int)toManage.UsedSizingPolicy & (int)
      SizingPolicy.ExpandXIfNeeded) > 0;
3
4 //update the center according to the new height and original top left (
      because that should stay the same)
5 if (overflowInX && expandXIfNeeded)
6 {
7     toManage.Width = x + xIncrease - topLeftX + 0.001;
8     toManage.CenterX = topLeftX + (toManage.Width / 2.0);
9 }
```

```
1  //called from the constructor of a container.
2  private void InitStyle()
3  {
4      //container is allowed to expand in x, and shrink in y
5      UsedSizingPolicy = SizingPolicy.ExpandYIfNeeded | SizingPolicy.
           ShrinkYIfNeeded;
6      MarginTop = (AlternativeIndex == 0) ? 0.3 : 0.0;
7      if (!Globals.ThisAddIn.Application.IsUndoingOrRedoing)
8      {
9          RShape.ContainerProperties.ResizeAsNeeded = 0;
10         ContainerPadding = 0;
11     }
12     LinePattern = 16;
13 }
```

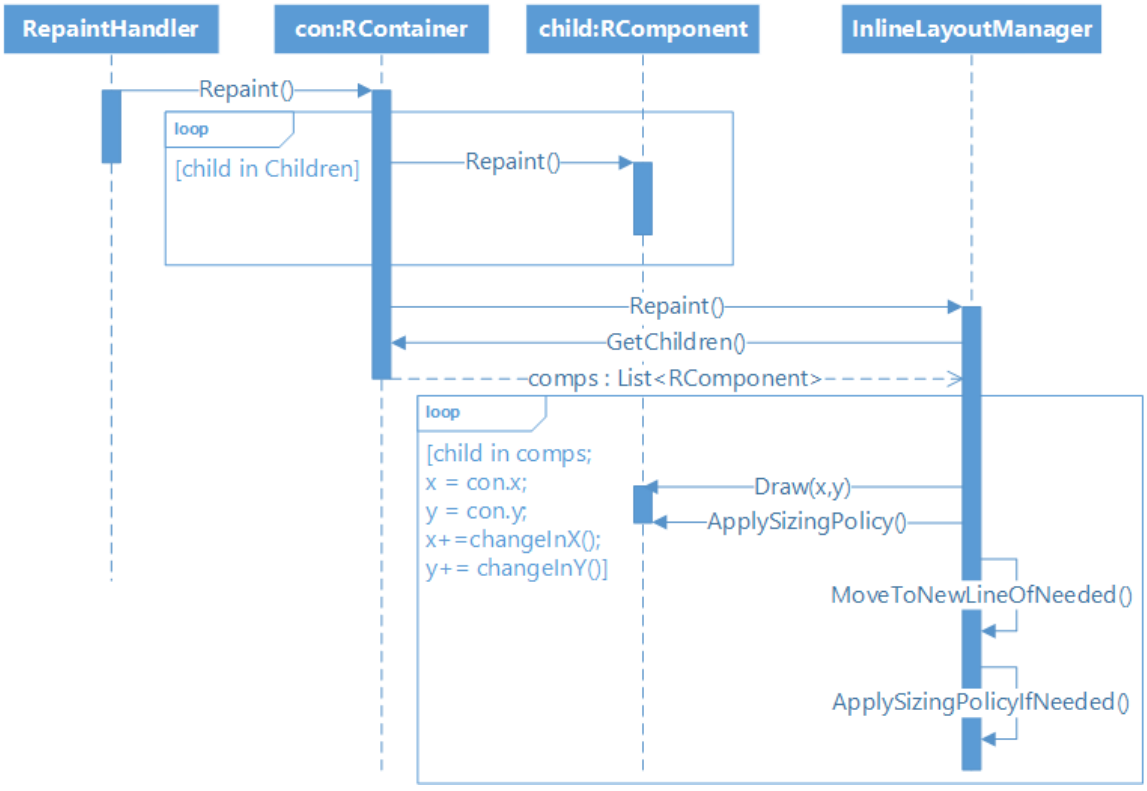

Figure 12: A simplified sequence diagram of the flow of layout managing in Rationally.

In Figure 12, the process that takes place during layout managing is visualised. Layout managers are called as part of a repaint event. Starting at the top of the View Tree (explained below), the layout managers of all components are recursively called. As can be seen in Figure 12, the container's children are repainted before the container itself. The container invokes its layout manager, which starts to place the children of the container in the container, taking into account the sizing policy of the container and the sizing policy of the child component. In the following sections, we will explain this process in more detail, starting with the inline layout manager and followed by the vertical stretch layout.

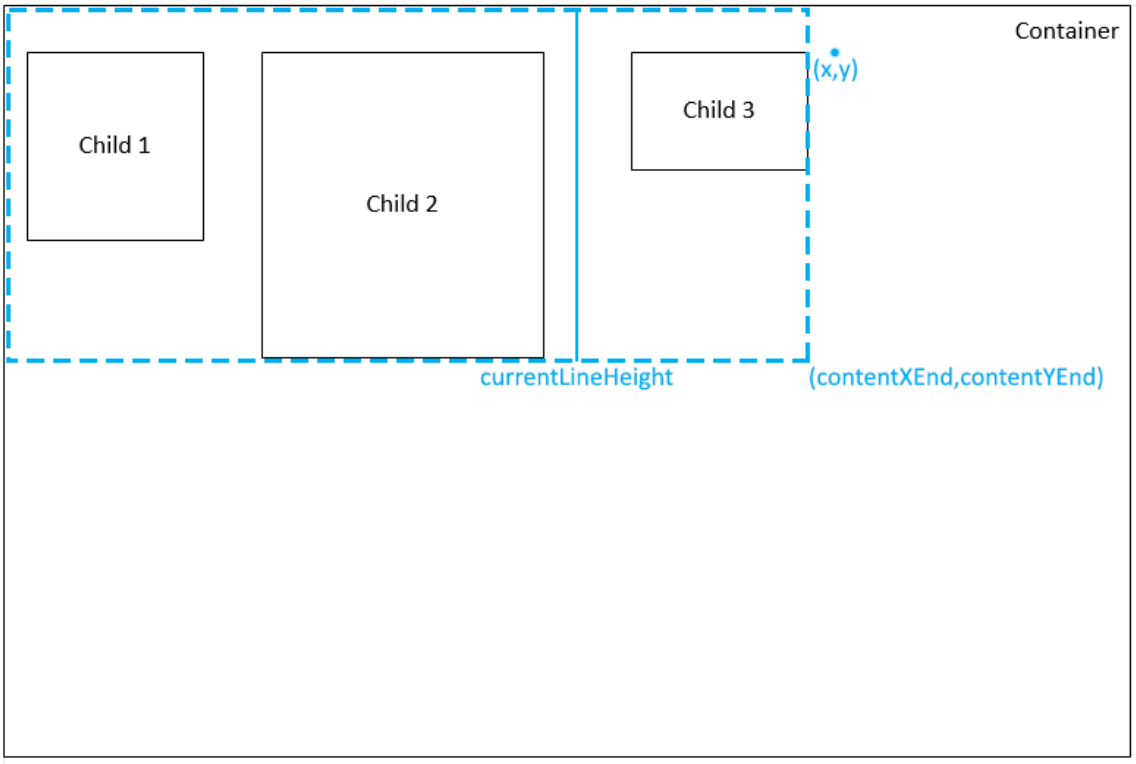

Figure 13: An example of the parameters involved during layout managing.

The *InlineLayout* manager's most important method is the *Draw* method. It recursively draws the first component in the *components* queue that is passed to it until the queue is empty. The x and y parameter point to the current drawing location of the manager (Figure 13). Because this manager tries to place elements horizontally next to each other, lines of components (that are next to each other) arise. The *currentLineHeight* contains the height of the line that the manager is in. In words, it is the highest height among the components that are next to each other within the current line. The parameters *contentXEnd* and *contentYEnd* store the most right x bound and lowest y bound of the content within the container. They define the smallest container from the left top of the container that wraps all laid out components.

The last three parameters (*currentLineHeight, contentXEnd, contentYEnd*) become relevant after all components have been drawn. Depending on the sizing policy of the managed container, they determine whether the container will shrink to the size of the smallest possible wrapper around the content. This can happen in zero, one or both dimensions. The line height comes into play when the container is not high enough for the last drawn line of components. The container might need to enlarge, if its sizing policy allows it, to the height of the last drawn line of components.

The *VerticalStretchLayout* has a simpler implementation. Where the *InlineLayout* stacks components horizontally as well as vertically, the *VerticalStretchLayout* only places the child components vertically below each other. The draw method only maintains the values of x and y. Because the y value is moved down to right below the last drawn component, it also stores the end of the content in the y dimension, after all components are drawn. Furthermore, the container does not adapt its width to its content, but rather it is the other way around.

A final difference between the two layout managers is the order of repainting between child and container. In our view tree, a repaint of parent first calls the *repaint* method on its children, before calling his own layout manager. Because the *VerticalStretchLayout* modifies (e.g. stretches) its children, it also calls the *repaint* method on its children after repainting itself, to allow the child to adapt its layout to its new size.

6.3 View Tree

To utilise the composite structure, we maintain a tree of *RComponents*, similar to the container/-component structure that Visio uses. For this we implemented behaviour related to adding and removing items from the tree. To allow for this, every *RComponent* has a secondary constructor that accepts a shape. Using this constructor, we can wrap existing shapes into a Rationally component, so that we may add it to the tree. For example, when a tree is rebuilt, the shape that is being added might be a *AlternativeContainer*. This shape is then thrown into the constructor for an *AlternativeContainer*, which uses fields from the shape to build the class.

In addition to this, every RComponent or child class has a method *AddToTree(Shape s)* that checks whether the shape that needs to be added is of a type that is a child of the current component. If so, it is added to the tree. For example, using the same *AlternativeContainer* as above, it is then added as child of an *Alternative**s**Container*. In the same vein, there is also a *RemoveFromTree(Shape s)* method, that removes the shape and its wrapper component.

This allows us to rebuild the tree at any point. However, we do not want items to be added to the tree multiple times. For that reason, we also implemented a *ExistsInTree(Shape s)* method that checks whether the shape exists in the tree and a *GetComponentByShape(Shape s)* method that also returns the component if it exists.

6.4 Eventhandler Registry

The most convinient way of communicating from an add-in with the Visio application is through events. That is, Visio will fire events when changes are made to the sheet and add-ins are allowed to register event handlers to those events on the Application object of Visio. One complication with Visio events is the fact that only one type of event can be fired outside the visio application, by add-in code: the marker event. Other events can only be listened to. When such a marker event is called, all registered event listeners are initiated, with only one context dependent parameter: the *contextString*. This parameter's value is decided by passing it in the creation of a marker event. This is done using the Visual Basic function *QUEUEMARKEREVENT(context)*. All the shapes that are programatically added to the sheet by Rationally come with a context menu. The only way to detect the fact that the user has clicked on one of these context menu, is by firing a marker event when it is clicked. This means that dozens of shapes with multiple context menu options all fire the same event, passing only a string as information. To avoid switch-statements and deep nesting in the handling of such a marker event, we created an event handler registry.

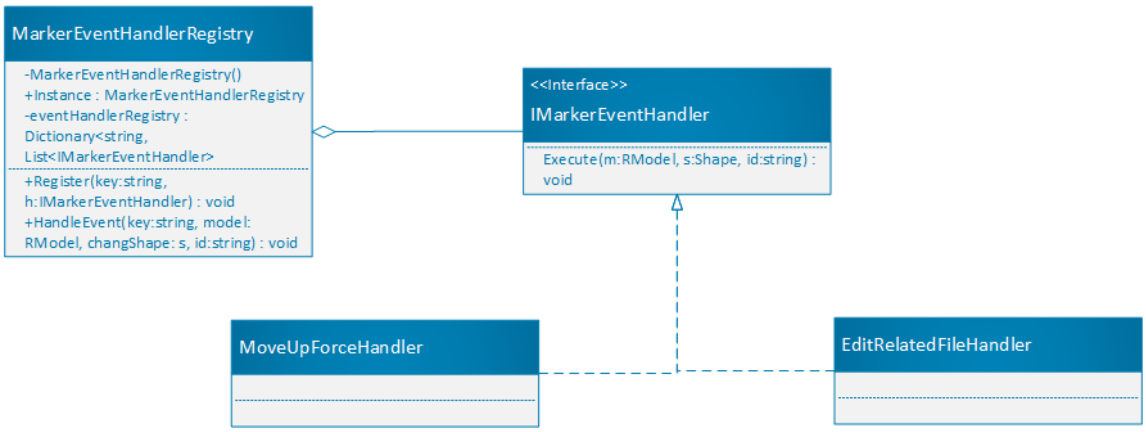

Figure 14: Class diagram of event handler registry and some related classes.

The event handler registry was implemented as follows. Firstly, as can be seen in Figure 14, we created a class for the registry, containing a mapping of event strings and event handlers. The type of this map is *Dictionary<List<string, IMarkerEventHandler>>*. This also supports a one-to-many mapping between event key and handler. The registry class also implements the singleton pattern, so all handlers must be registered at the same instance of the registry [8]. This is supported by the method *Register* (see Figure 14). Finally, an interface was created, containing a method definition that is implemented by all marker event handlers. This method will be executed by the registy on all the handlers registered on the event key that it receives as can be seen in the code fragment below.

```
1  //method implementation of the HandleEvent method, which is called when
       a marker event is fired in the application
2  public void HandleEvent(string eventKey, RModel model, Shape
       changedShape, string identifier)
3  {
4      if (Registry.ContainsKey(eventKey) && !Globals.ThisAddIn.
           Application.IsUndoingOrRedoing)
5      {
6          Registry[eventKey].ForEach(eh => eh.Execute(model, changedShape
               , identifier));
7      }
8      else
9      {
10         Console.WriteLine("NOTICE: marker event requested on key with
               to registered handlers: " + eventKey);
11     }
12 }
```

In the following piece of code, we register one of our methods in the *ThisAddIn* class as a listener for marker events. After that, a marker event handler is registered in the registry.

```
1  //register the listener on the application object
2  Application.MarkerEvent += Application_MarkerEvent;
3
4  //register an event handler
5  MarkerEventHandlerRegistry registry = MarkerEventHandlerRegistry.
       Instance;
6  registry.Register("alternatives.add", new AddAlternativeEventHandler())
       ;
```

Below, the (simplified) method implementation can be found of the method that was registered on the application object as a listener. A marker event is mostly triggered using menu options, implying that the shape of the menu is selected (otherwise, the menu was not available). We use this to find the shape whose menu was used and pass it along to the registered event handlers. This completes all the information required for all our marker event handlers.

```
1  private void Application_MarkerEvent(Application application, int
       sequence, string context)
2  {
3      foreach (Shape s in selection)
4      {
5          MarkerEventHandlerRegistry.Instance.HandleEvent(s.CellsU["User.
               rationallyType"].ResultStr["Value"] + "." + context, Model,
               s, identifier);
6      }
7  }
```

6.5 Undo/Redo behaviour

Visio offers the operation *undo* and *redo* to the user, respectively undoing and redoing the last modification that a user made to the sheet. These operations will update Visio's state, but not Rationally's model nor its view state. We needed a way to detect these actions when they occur and update Rationally's state accordingly. The operations on which *undo* and *redo* are available can be split up into the following categories:

1. Adding a (group of) shapes
2. Deleting a part of a group of shapes.
3. Deleting a container, that contains shapes.
4. Editing the state of a shape.

Besides these four categories, there are several methods of invoking these listed actions. These different methods of invocation come with different flows that should be accounted for.

1. Direct action (by pressing a key)
2. Context menu options
3. Composite invocation by selecting multiple shapes

When implementing undo/redo behavior, there are two main approaches to consider, the first of which is to maintain a stack of states. Every action that the user performs generates a new state to which the application can later be rolled back. The second approach is to invert operations at hand for every operation available to the user. When the user would want to undo an operation, the inverse operation would be invoked.

We decided to go with the second approach, because we already had the inverse operation at hand. For every add operation available on the sheet, for example, we also created a delete operation. We extracted these operations to seperate handlers, making them modular and easy to reuse as undo operations. We will now go over the different categories of undoable/redoable actions and describe our implementations for them.

The first category are formed by the add operations. Three of the main containers in our sheet offer add operations: *Alternatives*, *Evaluation* and *Related Documents*. These operations, when triggered, will add a shape to the respective container, which child shapes. When such an add-operations is undone, the shape and its child shapes are removed from the sheet. This fires a *ShapeDeleted* event, to which we listen in our add-in. The flow visualised in Figure 15 shows (somewhat simplified) what happens when an alternative addition is undone. The application raises a *shapeDeletedEvent* for every shape that will be deleted by this undo-action. Each shape is located in the view tree and,when found, removed from it together with its child components (using the recursive method *removeFromTree*). Important to note is that this does not lead to any shape deletion, because our view tree conists of RComonents that only wrap a Shape object and not contain one. After the update of the view tree, the delete handler will update the model by removing the deleted alternative from it. Finally, a *RepaintHandler* is created. This is required because the

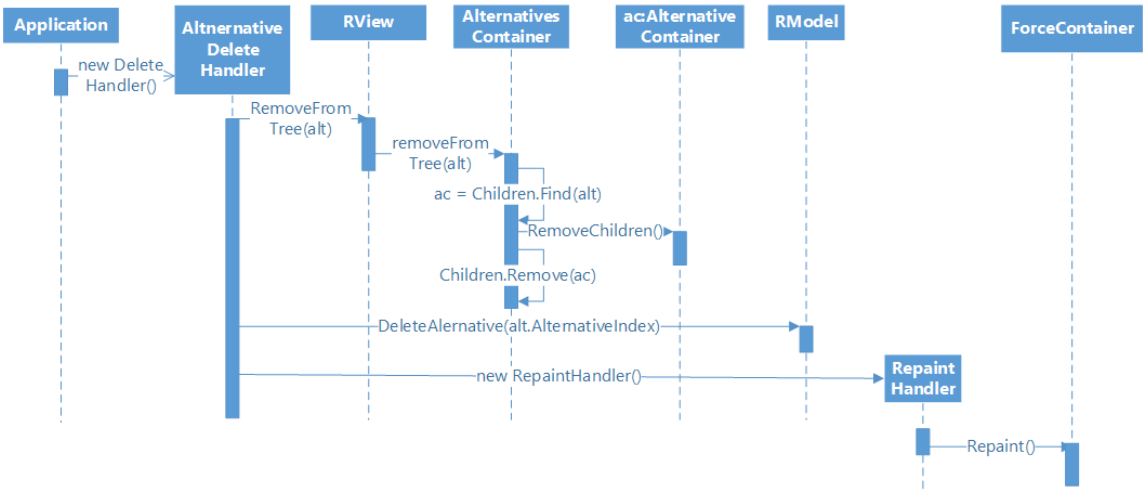

Figure 15: Sequence diagram of deleting an alternative.

forces table is dependent on the amount of alternatives and needs to be repainted.

We used this delete handler as the undo operation of an add-alternative operation. However, this comes with complications. The main problem with undo and redo in Visio add-ins can be phrased in one sentence:

while the application is undoing or redoing an operation of the user, shapes cannot be added, deleted, repositioned or resized.

Whenever we wanted to reuse a handler as an undo/redo handler, we had to consider this and therefore we implemented slightly different behaviour for those cases. To do so, we made use of the property *Application.IsUndoingOrRedoing* offered by the Visio API Application object. It indicates whether the application is in an undo/redo state. Our main approach in adapting handlers was wrapping prohibited statements in if-statements that made sure that application was not undoing or redoing, like the following code fragment.

```
1 if (!Globals.ThisAddIn.Application.IsUndoingOrRedoing) {
2     Globals.ThisAddIn.View.Children.ForEach(c => c.PlaceChildren());
3 }
```

The second category of operations is *Deleting a part of a group of shapes*. We will discuss this together with the third category, *Deleting a container that contains shapes.*. The undo/redo behaviour for these actions was implemented in a similar fashion as the add operation, by reusing the relevant add-handlers as undo/redo operations for the delete operations. The same constraints hold, and were taken into account in a similar way, but an extra complication arises beside them.

This extra complication can be explained as follows. When a child shape of an alternative (for example the title) is deleted, Rationally will delete the entire alternative (that is, the container, state, title, identifier and description). When this operation is undone, all these shapes are automatically readded to the sheet by Visio, but the order differs depending on what shape was deleted by the user. When the shapes are readded, a *ShapeAdded* event is raised that triggers our *AddToTree* method. It is responsible for readding the shape, wrapped in a correct Rationally class, to the view tree. It does the opposite of *DeleteFromTree*, but in the same recursive manner. The complication araises when a child shape is readded before its container, because it cannot be placed yet. However, it can neither be ignored and regenerated, because the specific shape might have been styled be the user.

We solved this complication by using stub components. These components are similar to RContainer objects in their interface, but the fundamental difference is that they do not require an RShape. The properties of the class that normally represent ShapeSheet properties are overloaded to fetch the value from a private field instead of the ShapeSheet of the RShape. Whenever a child-component arrives earlier than its container, we generate a stub container and insert it in the view tree before adding the child component to it. Once the container is added to the tree, the stub is recognised and replaced by the container.

The just described problem arises when a child component is being manually deleted by the user. Deleting the container first is just an easier procedure to undo or redo than deleting a child component first, so it did not result in additional problems.

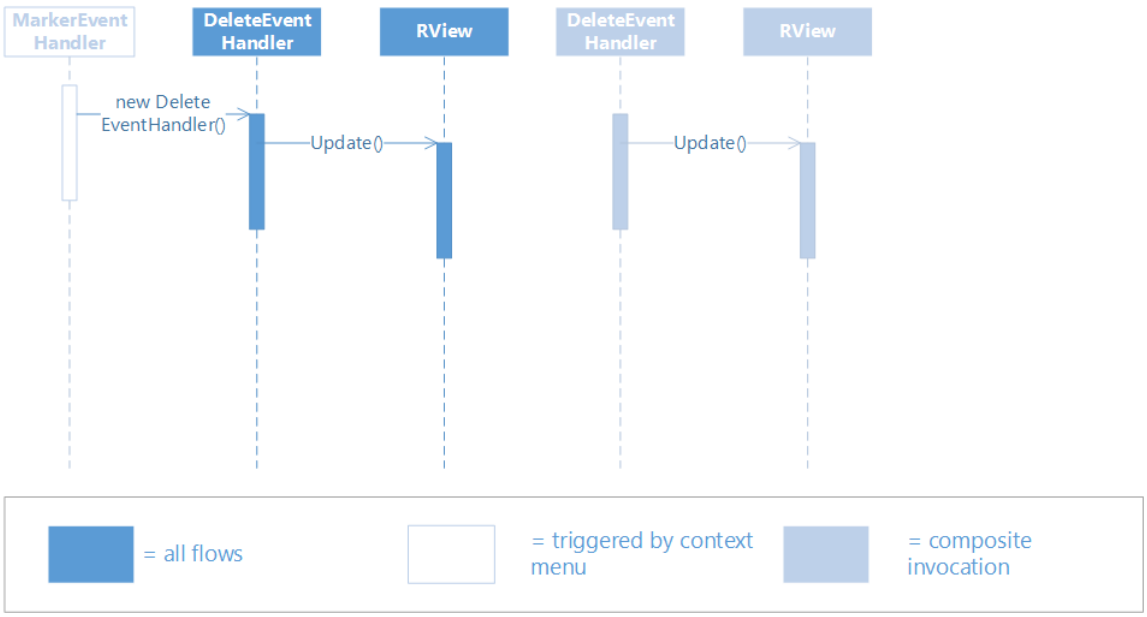

Figure 16: Sequence diagram showing different compositions for each flow.

The final undoable/redoable action is editing a component. We treated these events as regular edit events and were able to quickly implement the correct behaviour for it. Because Visio automatically resets the Shape to its previous state, we only had to update our model and did not require any prohibited operations.

The different methods of invocation were relatively easy to take into account, because the available operations are set up very modularly. The different methods can be seen as different flows, but when taking a closer look, they do not do anything different from others: they have a shared set of operations, that is extended is some of the flows. We treated each flow as a chain of action modules and were able to reuse a lot of modules using this approach. In Figure 16, the different required modules for each of the three flows are highlighted for a typical delete action. The manual detection only requires the execution of the delete handler. A deletion initiated through a context menu requires the same, but is preceded by a marker event handler which simply mimics a manual deletion. Finally, when the user selects multiple components, especially when the components are in different containers, the manual deletion is executed for each of the selected components. This concludes our support for undo/redo operations.

7 Evaluation

In this section evaluate the work we did on this project. We provide an overview of the requirements in combination with whether we succeeded in implementing them. If we did not, we provide an explanation for this. After that, we will reflect on our process during the development.

As can be seen in the table below, most requirements have been fullfilled, with only the deployment functionality missing and the performance not being up to par. However, even considering those things, we can still say that the application does what it is supposed to do very well. However, due to the missing deployment functionality, the application is not ready for the market. That functionality would need to be added first, before considering going into a beta phase on the market.

Now that we verified our technical requirements, we will look back at our working process, our planning and the tools we used during this project.

ID	Fullfilled	Explanation	ID	Fullfilled	Explanation
		Information			**Documents**
1.1	yes	–	5.1	yes	–
1.2	yes	–	5.2	yes	–
1.3	yes	–	5.3	yes	–
1.4	yes	–	5.4	yes	–
1.5	yes	–	5.5	yes	–
1.6	yes	–	5.6	yes	–
1.7	yes	–	5.7	yes	–
1.8	yes	–	5.8	yes	–
1.9	yes	–	5.9	yes	–
1.10	yes	–	5.10	yes	–
1.11	yes	–	5.11	yes	–
1.12	yes	–	5.12	yes	–
		Description			**Arguments**
2.1	yes	–	6.1	yes	–
2.2	yes	–	6.2	yes	–
2.3	yes	–	6.3	yes	–
		Alternatives			**History**
3.1	yes	–	7.1	yes	–
3.2	yes	–	7.2	yes	–
3.3	yes	–	7.3	yes	–
3.4	yes	–			**Stakeholders**
3.5	yes	–	8.1	yes	–
3.6	yes	–	8.2	yes	–
3.7	yes	–	8.3	yes	–
3.8	yes	–			**General**
3.9	yes	–	9.1	yes	–
3.10	yes	–	9.2	yes	–
		Evaluation	9.3	yes	–
4.1	yes	–	9.4	yes	–
4.2	yes	–	9.5	yes	–
4.3	yes	–			**Non-functional**
4.4	yes	–	10.1	yes	–
4.5	yes	–	10.2	yes	–
4.6	yes	–	10.3	yes	–
4.7	yes	–	10.4	yes	–
4.8	yes	–	10.5	no	Due to time constraints, no installation functionality is provided.
4.9	yes	–	10.6	no	Adding an alternative can take more than two seconds.
4.10	yes	–	10.7	yes	–
4.11	yes	–	10.8	yes	–
4.12	yes	–	10.9	yes	–
4.13	yes	–	10.10	yes	–

It was decided to use SCRUM as our development process, meaning that we created a backlog of user stories at the start of our project and that we implemented them in sprints. We had a weekly meeting in which we discussed our process and decided upon the schedule for the next sprint based on the results of the previous ones. The meetings were very constructive, in a way that they provided the details on the requirements we were going to implement the next sprint and that they provided us with feedback on our results of the previous meeting. The interval between meetings was long enough to make progress to discuss and short enough to not yield set backs, as a result of lack of communication. Wrong implementations were detected within seven days, meaning the consequences of those mistakes were never big.

At the start of the project, the idea was to split our project time into two halfs. The first half would be dedicated to the Visio add-in, the second half to setting up the server to store decisions and their relations. This turned out to be too optimistic, because the add-in was more work than expected. This extra work was caused by supporting undo and redo operations and understanding the Visio API, which is documented very poorly and sometimes even incorrectly, such as enumerations returning the wrong value. Halfway through the project, it was therefore decided to drop the server part from the scope of this project, giving us time to build a well functioning add-in, instead of two barely functioning applications. Not taking this decision into account, the planning for the project was good, since the add-in was finished in time and satisfies most requirements.

We used several tools during the development of this project. Firstly, we used Git as a code repository with version control. This worked well with Visual Studio, which we used to develop our add-in, because Git can be integrated in Visual Studio. Within Visual Studio, we used Resharper to automatically review our code. It suggests improvements on the code on various aspects, like accessability, logic simplifications and the detection of dead or useless code. We found that this helped us to keep our code quality at a high level from the beginning of our project and it saved us from having to refactor a lot in the final stages of the project. For our backlog and SCRUM board, we used Trello, which is a generic tool to maintain lists of cards. It suited our needs nicely and did not made our progress easy to keep track of.

All in all, we see that we implemented a large amount of our requirements. Our development process turned out the work well for us and our schedule was good. We did however not succeed in building the server that we wanted to integrate with the add-in. Finally, the development of the add-in went smoothly, because the tools we chose did what we wanted them to do.

8 Future Work

There are a few possible improvements to the application that should definitely be considered for in the future. In this section, we will discuss the addition of a server, how to expand the forces section, the addition of settings and the expansion to different Office products.

- **Addition of a lightweight server:** One of the biggest improvements would be the addition of a lightweight server on which decision documents could be stored. This would allow for the automation of certain components of the application, such as versioning or the changelog, to improve the Chronological viewpoint. It should also be possible to link decisions to each other on the server. This would allow for the Decision Relationship viewpoint to be implemented

and for the related documents section to be automatically generated.

- **Expanding the forces section:** The forces section also allows for quite a few improvements. One of these would be the exportation of the forces table to an Excel document, together with the names and states of the relevant alternative solutions. Currently, the forces table weighs every force the same way. However, a certain force such as performance might be way more important than the others. Therefore it would be a good improvement to allow for the weighing of forces in the table.
- **Colouring force values:** All force values are currently all coloured in the same way, independent of the height of a value. It would be useful to map the colours to the range of values, allowing for easier distinguishing of best solutions.
- **Settings:** There might also be a use for settings to be implemented. Names and colours for alternative states could be defined in these settings, instead of being defined by the application only.
- **Office products:** Another suggestion would be the expansion of the application to the other Office products, such as Word and Excel. This would allow for the documentation of different kinds of decisions in very different scenario's.

9 Conclusion

This project was undertaken to design a decision documentation tool for the embedded system industry. The tool was supposed to document these decisions using the viewpoint-based approach, which intends to document decisions from five viewpoints: the Decision Detail viewpoint, the Decision Relation viewpoint, the Decision Chronological viewpoint, the Decision Stakeholder viewpoint and the Decision Forces viewpoint.

The requirements at hand for Rationally were quite elaborate. Rationally was required to be built as an add-in for Visio, because of Visio's flexibility. It was also required to allow the documentation of all the five viewpoints that were described. Amongst the non-functional requirements user-friendliness was the most important one for this project. Rationally should add context to actions, but not constrain the user in his available actions.

We build our add-in using the Visio API that Microsoft created for this purpose. It offers interaction with Visio components like shapes, masters and stencils, but it also allows for direct interaction with the Visio application using events. This allowed us to create several interesting software components, like a composite structured component tree to maintain a state of the view. We also created layout managers for automatic positioning and sizing of shapes. One of the bigger challenges we overcame was implementing undo/redo operations on all our functionality, but it allowed us to reuse some of our existing components.

When we looked at the application we created considering our requirements, we found that we implemented most requirements, with only the deployment functionality missing and the performance not being up to par. The imperfect performance has no clear cause yet. It might be because our algorithms are far from optimal. However, since we are dealing with operations on relatively small

collections (10-100 elements), this is somewhat unlikely. The other potential cause would thus be more likely, which is that the large amount of interactions with the Visio application and shapes slows our add-in.

Although we did implement nearly all our requirements, there are still some possible extensions that would make our product even better. We could create a server that stores the relations of decisions, we could makes forces weighted, and finally settings could be implemented allowing for customizations.

In conclusion, this project has proven that it is possible to make a tool for the flexible embedded system industry that allows the documentation of decisions from all five described viewpoints, implemented as an add-in for a common tool.

References

[1] A. Tang, P. Avgeriou, A. Jansen, R. Capilla, and M. A. Babar, "A comparative study of architecture knowledge management tools," *The Journal of Systems and Software*, 2010.

[2] V. Clerc, P. Lago, and H. van Vliet, "Architectural knowledge management practices in agile global software development," *2011 IEEE Sixth International Conference on Global Software Engineering Workshop*, 2011.

[3] P. Kruchten, P. Lago, and H. van Vliet, "Building up and reasoning about architectural knowledge," *Lecture Notes in Computer Science*, 2006.

[4] R. Farenhorst and R. C. de Boer, *Knowledge Management in Software Architecture: State of the Art.* Springer, 2009.

[5] U. van Heesch, P. Avgeriou, and R. Hilliard, "A documentation framework for architecture decisions," *The Journal of Systems and Software*, 2011.

[6] U. van Heesch, P. Avgeriou, and R. Hilliard, "Forces on architecture decisions – a viewpoint," *Software Architecture (WICSA) and European Conference on Software Architecture (ECSA)*, 2012.

[7] C. Manteuffel, D. Tofan, P. Avgeriou, H. Koziolek, and T. Goldschmidt., "Decision architect - a decision documentation tool for industry," *The Journal of Systems and Software*, 2016.

[8] R. H. J. V. Erich Gamma, Ralph Johnson, *Design Patterns.* Addison-Wesley, 1994.